Out-of-Core Wavefront Computations with Reduced Synchronization

Pierre-Nicolas Clauss, Jens Gustedt, Frédéric Suter

Out-of-Core Wavefront Computations with Reduced Synchronization

Pierre-Nicolas Clauss*‡§ Jens Gustedt†‡§ Frédéric Suter*‡§

* Nancy University, France † INRIA, France ‡ LORIA § AlGorille

E-mail: {Pierre-Nicolas.Clauss,Jens.Gustedt,Frederic.Suter}@loria.fr

Abstract

Matrix computation algorithms often exhibit dependencies between neighboring elements inside loop nests such that the frontier between computed elements and those to be computed wanders in form of a 'wave' through the matrix. Macro-pipelining techniques can achieve an efficient parallelization of such algorithms by overlapping communication and computation. Usually these techniques are limited to situations where all the data to be processed fits into main memory, whereas for larger data the I/O usage pattern for external storage requires special attention. The work [5] presented a first extension of the wavefront framework to these so-called out-of-core problems. The present paper proposes a redesign of their algorithm that minimizes both overhead and perturbations coming from communications. To tackle the issue of non-contiguous I/O, we also propose an optimized data layout. These two major modifications of the original algorithm eventually allow us to present a third improvement as our implementation shortens the transition phase between two consecutive iterations of the wavefront algorithm. Experiments performed with the PARXXL *library show that we can significantly reduce the time lost during inefficient I/O operations and thus obtain faster computations.*

1 Introduction

Macro-pipelining methods arose in the context of parallel distributed memory computation with the goal to improve application performance and memory capacity, while still mastering the overhead due to communication. Unfortunately, classical improvement strategies for parallelization such as the choice of a good data distribution and the optimization of the communication layer to the lowest latency possible often do not show the desired speedups because of dependencies between computations and communications.

The two main ideas of macro-pipelining is to integrate the data flow dependencies into the algorithm design and to use asynchronous communications. Thereby, they allow for an overlap of computation and communication by reordering loops [8] and adding pipeline loops. These techniques can be used for several applications with wavefront computations like the ADI [9, 12], Gauss-Seidel [1], SOR [11], or Sweep3D [7, 16] algorithms.

For these techniques an immediate access to the data is crucial, but more and more applications express the need of computation on very large data sets, data much larger than what a conventional machine may store in main memory. Such computations then are strongly I/O bound because data must continuously be read, processed and written back to an external storage device, usually disk. They are referred to as out-of-core computations for which data is required to be loaded in a block-by-block pattern, where the block size has to be chosen so that the storage device access is improved. But even for the best block size, I/O may still be pretty slow because of a particular access pattern that an implementation might issue. Consider for instance data that is loaded with an inadequate pattern with respect to an I/O prefetching policy at system level, which usually expects data to be loaded in row-major order. Then each data block is fetched separately and the *latency* of the disk device dominates the I/O times. If on the other hand the data is loaded in the same order as the system fetches the blocks, the I/O times are only limited by the *bandwidth* of the device, and thus generally orders of magnitude faster.

A generic out-of-core wavefront algorithm in which communications and computations are overlapped by I/O have been proposed in [5] targeting distributed platforms. The original strategy using three different mem-

‡UMR 7503 CNRS - INPL - INRIA - Nancy 2 - UHP, Nancy 1

§INRIA project team with CNRS - INPL - Nancy 2 - UHP, Nancy 1

ory blocks in a cyclic way allows the overlap while saturating the disk resource. The distributed-memory implementation of such combination of macro-pipelining and out-of-core computing techniques allows to handle very large data but also shows several limitations that we propose to address in the present work. Indeed part of the performance study of [5] investigates the origins of the overhead of the implementation with regard to a simple read/write of the data. For the most favorable block size, the overhead can be decomposed into three equivalent parts. One third is induced by the non-contiguous read of a specific part of the data, another third comes from communication costs between neighbors while loading the pipeline and the final third reflects the perturbations induced by the asynchronous communications in the steady-state phase of the pipeline.

In this paper we propose to redesign the algorithm of [5] in order to reduce the synchronization overhead and address the aforementioned issues. For a simplification of the arguments and the implementation, we will describe our algorithms in terms of shared-memory. This redesign will allow us to minimize both overhead and perturbations coming from communications. To tackle the issue of non-contiguous I/O, we also propose an optimized data layout. These two major modifications of the original algorithm eventually allow us to present a third improvement as our implementation shortens the transition phase between two consecutive iterations of the wavefront algorithm.

In Section 2, we summarize existing related work. In Section 3, we describe the application that we use as an illustration of our algorithm design modifications. Section 4 details our optimized data layout while Section 5 addresses communication and synchronization issues. Section 6 shows experimental results for the optimized layout and the Livermore loop 23 on a Grid'5000[1] node. We conclude and present some future work in Section 7.

2 Related Work

The work presented in this paper is related to several research fields in high performance computing. First it considers a specific class of parallel algorithms that use macro-pipelining techniques to exhibit parallelism in matrix computations. Models and implementations of such algorithms have been proposed both for distributed memory [1, 7, 9, 11, 12, 16] and shared memory machines [2]. But these works focus on data that fit into memory. The present work addresses the issue of out-of-core computation in which the size of the data needed to perform a given computation exceeds the memory capacity of the computing platform. To efficiently handle such large data, two approaches can be followed. A system approach will modify the virtual memory manager of the operating system [4] or act on the file system [3] to reduce the impact of loading data from secondary storage devices.

In [15] the authors propose a programming framework to ease the implementation of processing pipelines on out-of-core data. But the most connected work is [5] in which an implementation of a pipeline algorithm on out-of-core data is proposed. It relies on an original strategy using several memory blocks accessed in a cyclic way to overlap computation, communication, and I/O and thus achieve a saturation of the disk resource which is the bottleneck. This work targets distributed memory clusters.

The originality of the present work with regard to [5] is twofold: First, we introduce an optimized data layout to decrease the time needed to load data from disk. This kind of technique has been successfully used in [13, 14, 17]. Then, we relax the synchronization constraints between the processors to improve the communication/computation overlap.

Another difference from previous experimental work in this context is that our implementation does not use raw file-I/O (POSIX `read/write`) to access the external storage device, but the performing file-mapping facilities (`mmap`) in combination with the normalized memory access advisory interface (`posix_madvise`). Portable and efficient access to these system features is provided through the PARXXL library [6]. PARXXL handles data through `chunks`. These objects indirectly represent the data which may be given as disk files, stack memory or POSIX' shared segments. A `chunk` then controls when, how and how much data is mapped into the address space of the process and provides means to access this data in RAM.

3 The Target Application

We applied our approach to a wavefront algorithm, the Livermore loop 23 [10], which corresponds to the computation of an implicit hydrodynamics kernel. This algorithm is one of the Livermore loops which is part of the LinPack benchmark collection. This is one of the benchmarks used by the TOP500 to classify the 500 most powerful computers in the world. This algorithm is applied on every element of a 2D-matrix with a depen-

[1] `https://www.grid5000.fr`

dency to four neighbors: North, South, East and West. The algorithm is then applied again until convergence is reached, which typically occurs after several dozens of iterations.

The left part of Figure 1 presents the pseudo-code of the Livermore loop 23 for a single iteration while the right part of this figure shows the steps of the update of a single element of the data matrix. Elements of five coefficient matrices (`zb`, `zu`, `zv`, `zr` and `zz`), denoted by white rectangles labeled by matrix name and *row*, *column* indices, are needed by the computation. Four of them are multiplied (*1*) by the neighbors of the currently computed element. Two neighbors (North and West, in light gray) have already been updated and the other two (South and East, in dark gray) will be updated by further iterations. The results are then summed (*2*) with the fifth coefficient and the current element value is subtracted (*3*). This result is eventually multiplied by a constant (*4*) before being added to the current element (*5*).

The algorithm of the Livermore loop 23 is similar to most wavefront algorithms, as it shows data dependencies in the four directions. This allows the use of a pipeline-aware slicing of the data to achieve parallel computations. Still, this algorithm has its own specificity as it does not only rely on a single data matrix, but also on five coefficient matrices. This means that for a specific data matrix size, the total data memory footprint is actually six times bigger. Consequently data matrices handled in this paper are in-core as they fit into memory but applying this particular algorithm on such data remains an out-of-core computation because of the coefficient matrices.

4 An Optimized Data Layout

In this section we propose an optimized layout designed to improve I/O operations relying on a classic data distribution. Figure 2 shows what corresponds to the different notations used in this paper to describe the Livermore loop 23 algorithm when data are distributed following a row-major distribution on a ring of P processors. Each processor owns M/P rows of the data. Each row contains N elements. In case of an out-of-core data, this $(M/P) \times N$ block is stored on disk.

The right part of Figure 2 describes which part of the $(M/P) \times N$ block of data stored on disk is actually loaded in memory. A coarse grain wavefront approach will divide the computation of an iteration in $\lceil N/NB \rceil$ steps. A $MB \times NB$ block will thus only be loaded in memory at a given moment. Therefore four parts of this block have to be distinguished. Two of them are needed

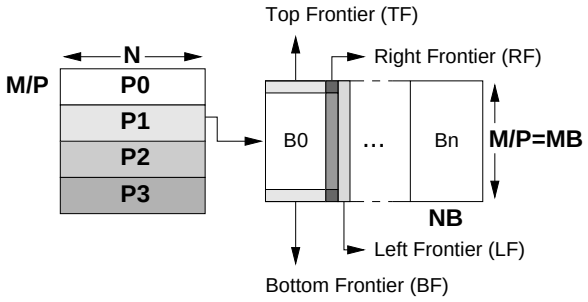


Figure 2. Notations used in wavefront algorithms.

before the beginning of a block update. Processors P_1 to P_{P-1} have to send the first row of a block to their left neighbor in the ring. We denote this row as the *Top Frontier (TF)*. The first column of the next block to update, denoted as *Left Frontier (LF)*, is also needed. The other two special parts of a block correspond to updated data that cannot be written on disk at the end of the step. The first one, denoted as *Right Frontier* (RF), is needed to compute the next step of the current iteration on the same processor. The second one is the *Bottom Frontier* (BF), that has to be sent to the right neighbor to allow it to update its data.

A classic row-major data layout, as shown by Figure 3(a), can cause a significant overhead as reading an entire block or a single vertical frontier implies several small reads and disk head moves. Our idea is to change the matrix representation so that loading a block or any single frontier can be done in only one read operation. This is done by changing the order in which elements of the matrix are stored on disk. In this new representation, we store the elements of TF first, then we store elements of LF, elements that are not in any frontier, elements of RF and finally elements of BF. Figure 3(b) shows this transformation for a $n \times n$ block (top indices are rows and bottom indices are columns). This new optimized data layout duplicates the four elements located at the corners of the block, but allows to read the entire block or any frontier in a single read operation, as elements are stored contiguously on disk. This reading pattern gives much better performance because it is compliant with the system's prefetching policy. Indeed the very common system policy expect data to be read linearly and thus performs readahead prefetching according to that policy. The fact that our optimized layout allows any kind of read (frontiers and blocks) linearly helps to get the best of the system prefetching features.

```
for i = 2, N − 1
  for j = 2, M − 1
    q = data[i−1][j] · zb[i][j]
      + data[i][j−1] · zv[i][j]
      + data[i][j+1] · zu[i][j]
      + data[i+1][j] · zr[i][j]
      + zz[i][j]
      − data[i][j]
    data[i][j] += 0.175 · q
  done
done
```

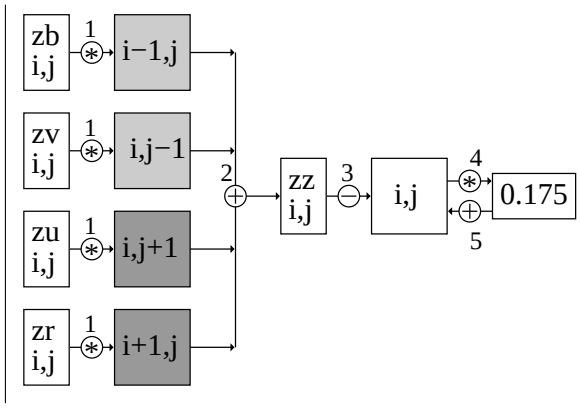
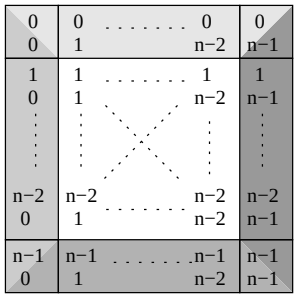
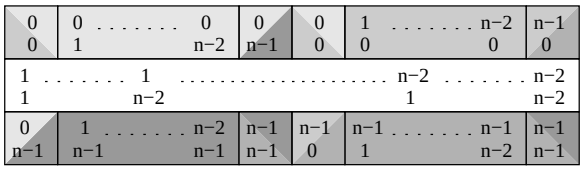


Figure 1. Livermore loop 23 algorithm.



(a) Row-major.



(b) Optimized.

Figure 3. Row-major and Optimized Data Layouts.

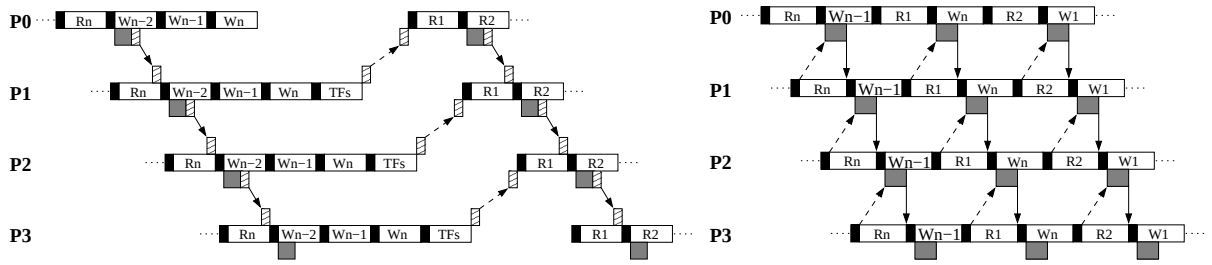
Block size	Space overhead	
4096×4096	512 Bytes	2.38e-05 %
512×512	32 KiB	0.0015 %
64×64	2 MiB	0.098 %
8×8	128 MiB	6.25 %

Table 1. Space overhead, 16384×16384 matrix of `double` (2 GiB).

Although the initial rewriting we propose implies the duplication of some data, the extra storage space needed is negligible compared to the total storage space used by the entire matrix. For an $N \times M$ matrix divided into blocks of size $NB \times MB$, the space overhead for an algorithm involving block frontiers of thickness T is $4 \cdot T^2 \cdot sizeof(element) \cdot N/NB \cdot M/MB$ bytes. In the particular case of the Livermore loop 23, the thickness is equal to 1. Table 1 shows the space overhead for a 2 GiB matrix depending on the block size. For the larger block sizes that turned out to be the most efficient in the experiments (see below) this memory overhead is clearly negligible. But we can see that even for very small blocks (8x8), the induced space overhead is low (6.25%).

A similar layout is used for the coefficient matrices, but as the algorithm does not need to transfer frontiers for these matrices, the layout simply consists in storing the elements block-wise instead of row-major, thus allowing a complete block load in a single read operation. This layout is called *block layout*. As we will show in Section 6, the use of a row-major layout strongly degrades the performance of the macro-pipeline and the block layout is sometimes used instead. We will thus compare our optimized layout to the block layout both in terms of matrix rewriting and computation time.

Figure 4 shows the processing time needed for switching from row-major layout to both block layout and our optimized layout. This operation is referred to as *preprocessing*. The figure also shows the reverse operation referred to as *postprocessing*. The time required to perform these operations appears to be very low and almost the same for all kind of operations, except preprocessing the optimized layout. The time overhead for



(a) Flushing the pipeline between iterations

(b) No pipeline flush between iterations

Figure 5. The impact on the communication/computation overlap.

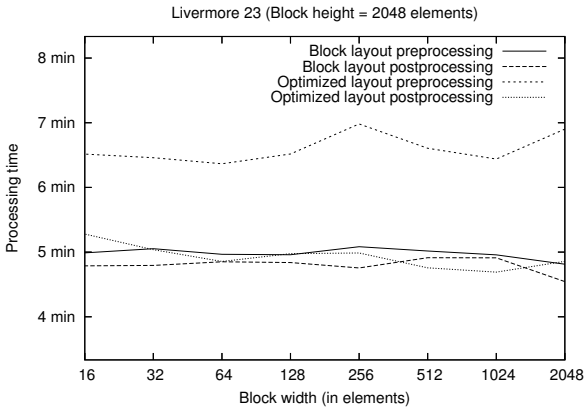


Figure 4. Layout preprocessing for block and optimized layouts, 16384×16384 **matrix of** `double`**.**

processors in both ways: the first processor needs the second's TF to start computing and the second needs the first's BF after it has been computed, as shown in Figure 6.

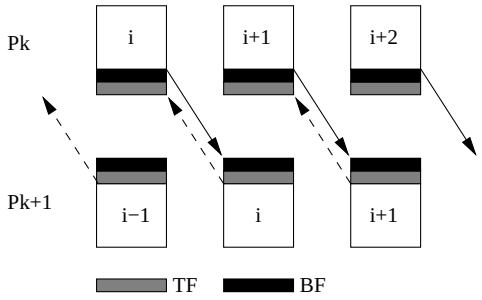


Figure 6. Communication pattern for wavefront algorithms.

this processing is due to the particular I/O pattern which is the consequence of reading vertical frontiers (*LF* and *RF*) in a row-major layout matrix.

The preprocessing and postprocessing operations are only required to be performed before and after running the application, if it is designed to work with the target layout. As the application is an iterative process, the time gained from using a better layout largely compensates the time spent to reshape the matrix to and from this layout, when compared to the overall time of computation with a row-major layout and no pre/postprocessing at all.

5 Improved Communications

The data dependency of the Livermore loop 23 algorithm requires that data is transferred between a pair of

This communication pattern requires to communicate data in both ways. To avoid network issues, communication in [5] has been done half-duplex by first sending each processor's entire first row of data to its left neighbor in the processor ring. This row of data regroups all TFs the previous processor will need, thus allowing only BFs to be sent between two processors during an iteration. This communication scheme, as shown in Figure 5(a), induces a flush state at the end of the pipeline (when all processors need to send their entire row) and a loss of time waiting for the pipeline to be refilled.

Our implementation of the Livermore Loop 23 algorithm aims at reducing the impact of the communication overhead on total processing time. Therefore we have to ensure that the data is communicated asynchronously once it is ready, in particular every TF is made available to the target once the corresponding block has been computed. It is no longer necessary to communicate the en-

tire first row of data at once, and thus the target processor does not have to wait for that entire row to start the next iteration of the outer loop. In our shared memory implementation, as all processors are sharing the same address space, communication thus reduces to filling some memory and then allowing the recipient to read the data, as shown in Figure 5(b). This is realized by the data abstraction of PARXXL called `chunks`, see Section 2. In a shared memory context this facility of PARXXL allows for a direct (but controlled) mutual access of the processes or threads to their respective data. Sending and receiving data is done by writing (resp. reading) some `chunk` and notification is done by a local rendez-vous between the sender and the recipient.

6 Experimental Results

The new communication pattern allows continuous computation without flushing the pipeline at the end of an iteration. It is thus possible for each processor to start a new iteration immediately after the end of the previous one. Figure 8 shows such a situation, where processors P_0 and P_1 have started the $n + 1^{th}$ iteration while processors P_2 to P_5 are finishing the n^{th} iteration. This situation we called *wave chaining* is as if two waves were traversing the matrix at once. By increasing the number of computing threads per processors it would be possible to increase the number of simultaneous waves even more. On the other hand, more blocks would reside in memory simultaneously and thus the block size would have to be tuned differently.

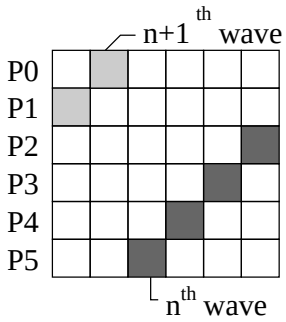


Figure 8. Wave chaining: two waves are traversing the matrix at once.

We implemented the Livermore Loop 23 algorithm using block layout and optimized layout, and the PARXXL library. In particular, the use of file mapping for accessing data allows very good I/O performance. The library also proposes very useful features, including data extraction (used for frontiers purpose) and shared memory communication with cross-pointers and local rendez-vous.

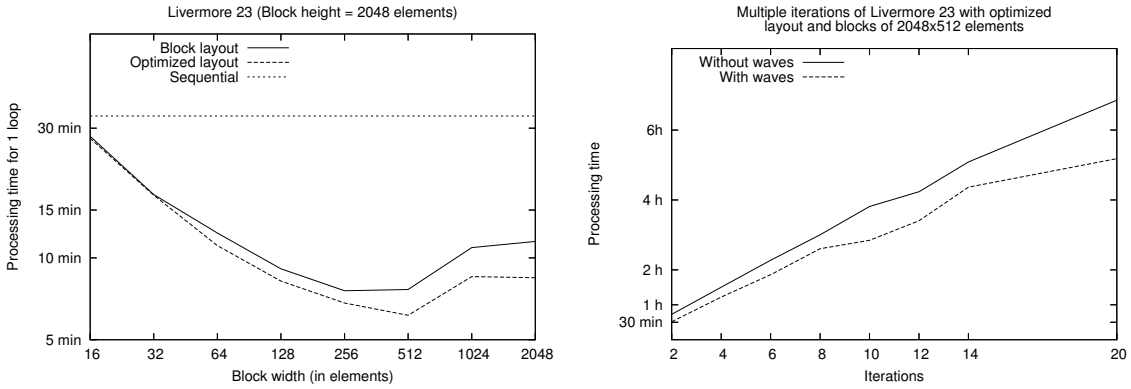
Figure 7(a) shows the execution time for one iteration of the algorithm on a 16384×16384 matrix of double precision elements for each of these implementations. Such a matrix has $2^{14} \times 2^{14} \times 2^3 = 2^{31}$ bytes (2 GiB) and the whole problem consisting of 6 such matrices (12 GiB) did not fit into the (2 GiB) RAM of our bi-processor (each with two cores) target machine.

This figure also represents the execution time for one iteration of the sequential algorithm. In this particular implementation, there is no block concept, so we represented for all block sizes subsequently tested with the other implementations the value of the sequential execution. This value is the upper limit above which the use of parallelism fails to improve performance. For example, applying macro-pipelining on a row-major matrix leads to performance at best ten times worse than the sequential execution time. The poor results observed in this case show that using a more adapted layout is the key to performance.

For the two tested layouts, the figure shows that there is a block size which gives better results. This situation is common to pipelined algorithms for out-of-core problems, where the I/O device bounds the performance for both very big and very small blocks because of disk bandwidth and/or latency.

For the best case, we can observe that the use of the optimized layout gives a 20% improvement over the block layout, for a single iteration. This is repeated for each iteration of the wavefront algorithm, which leads to an overall significant performance gain.

Figure 7(b) shows that wave chaining also brings better performance. The avoidance of flushing state allows iterations to start and finish earlier. This leads to another performance gain of 20%. But Figure 7(b) also shows that the average processing time for one iteration is higher when performing more than one iteration at a time. This may be due to the impact of using a single disk for storing data. In fact, when a new wave is started, part of the processes require data to be accessed at the beginning (B) of the file while the rest of the processes require data to be accessed at the end (E) of the file. These accesses are performed by threads running in parallel and thus can occur interleaved in any possible way. If the interleaving pattern is an alternating B-E, then the disk head spends most of the time to go back and forth in the file, which can lead to a severe loss of time per iteration. This may be avoided by increasing



(a) Execution time for one iteration

(b) Execution time for multiple iterations

Figure 7. Execution times for matrices of 16384×16384 `doubles`.

the number of disks and slicing data to be stored across the disks. This may allow a new wave to start accessing its data on another disk than data accessed by the current wave.

7 Conclusion and Future Works

In this paper we addressed the issue of parallelizing a wavefront algorithm on large matrices that do not fit into memory. We focus on a well known algorithm, the Livermore loop 23, which is part of the LinPack benchmark suite. To reduce the unavoidable I/O overhead of such an out-of-core computation we proposed an optimized data layout, at the cost of a negligible space overhead, thereby allowing the I/O operations needed by the algorithm to be contiguous. The use of the PARXXL library to access data through advanced mapping also helps to get more efficient I/O operations.

We compared our optimized data layout to the more common block layout, which is generally used to replace original row-major layouts. Our experimental results show that using our optimized data layout improves performance by 20% for the best block-size case.

Thus our goal of providing a better execution environment by combining an optimized data layout with advanced mapping features is fully reached. Besides these improvements, we think that it is possible to get even better results by increasing the number of storage devices, especially for shared-memory architectures. This should help to open up the bottleneck consisting of one single hard disk that is used for multiple accesses to the same file and would allow efficient wave chaining.

We expect our data model to scale to other architectures, including supercomputers and distributed environments, both at cluster and grid level. Since this data model is not algorithm or problem dependent, it also makes an automatic data distribution mechanism possible. This would then simplify the algorithm design for a lot of similar problems. A cellular automaton, based on some algorithm description (including data and block sizes, frontier thickness,...) could describe the data dependency and a problem specific subroutine would be used to attack each individual block.

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see `https://www.grid5000.fr`).

References

[1] S. Baden and S. J. Fink. Communication Overlap in Multi-tier Parallel Algorithms. In *Proceedings of SuperComputing'98*, Orlando, FL, Nov. 1998.

[2] K. Balasubramanian and D. Lowenthal. Efficient Support for Pipelining in Software Distributed Shared Memory Systems. *Parallel and Distributed Computing Practices*, 4(2), 2001.

[3] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A Model and Compilation Strategy for Out-of-Core Data Parallel Programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, Santa Barbara, CA, 1995.

[4] E. Caron, O. Cozette, D. Lazure, and G. Utard. Virtual Memory Management in Data Parallel Applications. In *Proceedings of High-Performance Computing and Networking, 7th International Conference, HPCN Europe 1999*, volume 1593 of *Lecture Notes in Computer Science*, pages 1107–1116, Amsterdam, The Netherlands, Apr. 1999. Springer.

[5] E. Caron, F. Desprez, and F. Suter. Out-of-Core and Pipeline Techniques for Wavefront Algorithms. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, CO, Apr. 2005. IEEE Computer Society.

[6] J. Gustedt, S. Vialle, and A. De Vivo. parXXL: A fine grained development environment on coarse grained architectures. In B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Workshop on state-of-the-art in scientific and parallel computing (PARA'06)*, 2006.

[7] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proceedings of the 2000 International Conference on Parallel Processing (ICPP'00)*, pages 219–232, Toronto, Aug. 2000.

[8] K. Ishizaki, H. Komatsu, and T. Nakatani. A Loop Transformation Algorithm for Communication Overlapping. *International Journal of Parallel Programming*, 28(2):135–154, 2000.

[9] D. Lowenthal. Accurately Selecting Block Size at Runtime in Pipelined Parallel Programs. *International Journal of Parallel Programming*, 28(3):245–274, 2000.

[10] F. McMahon. The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[11] T. Nguyen, M. Mills Strout, L. Carter, and J. Ferrante. Asynchronous Dynamic Load Balancing of Tiles. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, Mar. 1999.

[12] D. Palermo. *Compiler Techniques for Optimizing Communications and Data-distribution for Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[13] P. J. Rhodes, X. Tang, R. D. Bergeron, and T. M. Sparr. Iteration Aware Prefetching for Large Multidimensional Datasets. In *Proceedings of the 17th International Conference on Scientific and Statistical Database Management (SSDBM 2005)*, pages 45–54, Santa Barbara, CA, June 2005.

[14] P. J. Rhodes, X. Tang, R. D. Bergeron, and T. M. Sparr. Out-of-core Visualization Using Iterator-Aware Multidimensional Prefetching. In *Proceedings of Visualization and Data Analysis 2005*, volume 5669, pages 295–306, San Jose, CA, Mar. 2005. SPIE.

[15] E. Riccio Davidson and T. Cormen. Building on a Framework: Using FG for More Flexibility and Improved Performance in Parallel Programs. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CO, Apr. 2005. IEEE Computer Society.

[16] D. Sundaram-Stukel and M. K. Vernon. Predictive Analysis of a Wavefront Application Using LogGP. *ACM SIGPLAN Notices*, 34(8):141–150, 1999.

[17] W. Zhou and D. Lowenthal. A Parallel, Out-of-Core Algorithm for RNA Secondary Structure Prediction. In *Proceedings of the 35th International Conference on Parallel Processing (ICPP'06)*, Chicago, IL, Aug. 2006.